《程序设计与算法》——《C 程序设计进阶》学习笔记

Du Ang
du2ang233@gmail.com

2017 年 8 月 10 日

1 《C 程序设计进阶》课程的主要内容

- 结构化的程序——函数
  - 函数
  - 递归
- 更多的数据结构
  - 指针
  - 结构体
  - 链表

2 C 程序中的函数

2.1 C 程序中常用的函数

- `r = sqrt(100.0);`
- `k = pow(x, y);`
- `i = strlen(str1);`
- `v = strcmp(str1, str2);`
- `n = atoi(str1);`

2.2 函数的调用方式

1. 作为独立语句 例如 `stringPrint();`，调用函数完成某项功能，没有任何的返回值。
2. 作为表达式的一部分例如 `number = max(numA, numB) / 2;`
3. 以实参形式出现在其他函数的调用中例如 `min(sum(-5, 100), numC);`

2.3 函数是 C 程序的基本单位

`main()` 函数是程序执行的入口，新的 C 语言标准要求 `main()` 函数需要有 `int` 类型的返回值。
函数的类型是指“函数返回值的数据类型”。
函数的原型 = 返回值类型 + 函数名 + 参数类型

2.4 参数的传递

实参和形参具有不同的存储单元，实参与形参变量的数据传递是“值传递”。函数调用时，系统给形参分配存储单元，并将实参对应的值传递给形参。实参与形参的类型必须相同或可以兼容。

2.5 数组与函数

注意数组元素和数组名作为函数参数时的区别。

```
void changeByElement(int a, int b)
{
    a = 30;
    b = 50;
}

void changeByArrayName(int a[])
{
    a[0] = 30;
    a[1] = 50;
}

int main(int argc, char const *argv[])
{
    int a[2] = {3, 5};
    changeByElement(a[0], a[1]);
    cout << a[0] << " " << a[1] << endl;    // 3 5
    a[0] = 3;
    a[1] = 5;
    changeByArrayName(a);
    cout << a[0] << " " << a[1] << endl;    // 30 50
    return 0;
}
```

数组名不是变量，而是数组在内存中的一个地址，是一个常量。

2.6 局部变量和全局变量

局部变量 在函数内或块内定义，只在这个函数或块内起作用的变量。

全局变量 在所有函数外定义的变量，它的作用域是从定义变量的位置到本程序文件的结束。
全局变量的缺点：

- 破坏了函数的“相对独立性”
- 增加了函数之间的“耦合性”
- 函数之间的交互不够清晰

不在非常必要的情况下，不要使用全局变量。

3 Assignment 1

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    std::cout << std::setfill('x') << std::setw(10);
    std::cout << 77 << std::endl;   // xxxxxxxx77
    std::cout << std::setfill('x') << std::setw(10) << right;
    std::cout << 77 << std::endl;   // xxxxxxxx77
    std::cout << std::setfill('x') << std::setw(10) << left;
    std::cout << 77 << std::endl;   // 77xxxxxxxx
    return 0;
}
```

4 函数的递归

函数不能嵌套定义，所有函数一律平等。
函数可以嵌套调用，无论嵌套多少层，原理都一样。

4.1 用递归来完成递推

递归和递推的不同点：

- 递推的关注点放在起始条件上
- 递归的关注点放在求解目标上

递归和递推的相同点：

- 都重在表现第 i 次和第 i+1 次的关系

用递归实现递推的典型案例：斐波那契数列，进制转换
用递归实现递推的优点：让程序变得简明。
用递归实现递推的方法：

- 把关注点放在要求解的目标上
- 找到第 n 次做与第 n-1 次做之间的关系
- 确定第 1 次的返回结果

4.2 用递归模拟连续发生的动作

典型案例：汉诺塔问题
用递归模拟连续发生的动作的方法：

- 搞清楚连续发生的动作是什么

```
void move(int m, char x, char y, char z);
```

- 搞清楚不同次动作之间的关系

```
move(m - 1, x, z, y);
cout << "Move one plate from " << x << " to " << z << endl;
move(m - 1, y, x, z);
```

- 搞清楚边界条件是什么

```
if (m == 1) {
    cout << "Move one plate from " << x << " to " << z endl;
}
```

4.3 用递归自动分析

典型案例：逆波兰表达式，放苹果问题
用递归自动分析的方法：

- 先假设有一个函数能给出答案

```
count(int m, int n)
```

- 在利用这个函数的前提下，分析如何解决问题

```
if (m < n)
    return count(m, m);
else
    return count(m, n-1) + count(m-n, n);
```

- 搞清楚最简单的情况下的答案是什么

```
if (m <= 1 || n <= 1)
    return 1;
```

5 指针

5.1 什么是指针

把某个变量的地址称为“指向该变量的指针”。

可以通过取地址运算符 `&` 来取到一个变量的地址。

计算机通过变量的地址（指针）操作变量，利用指针运算符 `*`。`*&c` 等价于 `c`。在编译时，编译器会建立变量名到地址的映射。

指针变量是专门用于存放指针（某个变量的地址）的变量。

定义一个指针变量：`int *pointer;`

其中，`int` 是指针变量的基类型（指针变量指向的变量的类型），`*` 是指针运算符，表明了 `pointer` 的类型，`pointer` 是指针变量的名字。

```
int c = 76;
int *pointer;    // 定义名字为 pointer 的指针变量
pointer = &c;    // 将变量 c 的地址赋给 pointer。赋值后，pointer 指向 c
// pointer = c; // 错误！pointer 只能存放地址！
```

指针变量也是变量，是变量就有地址。

5.2 & 与 * 的运算优先级

后置 ++/- - > 前置 ++/- -、逻辑非（!）、*、& > 算术运算符 > 关系运算符 > &&、|| > 赋值运算符

- `&*pointer = &(*pointer)`
- `*&a= *(&a)`
- `(*pointer)++` 不等于 `*pointer++`

5.3 pointer++ 的含义

```
#include <iostream>
using namespace std;
int main() {
    int n = 0;
    int *p = &n;
    cout << p << endl;  // 00C6FED8
    p++;
    cout << p << endl;  // 00C6FEDC
    return 0;
}
```

假设：`iPtr` 当前所存的地址是 0x00000100

- 若 `iPtr` 指向一个整型元素（占四个字节），则 `iPtr++` 等于 `iPtr + 1 * 4 = 0x00000104`

- 若 iPtr 指向一个实型元素（占四个字节），则 iPtr++ 等于 iPtr + 1 * 4 = 0x00000104
- 若 iPtr 指向一个字符元素（占一个字节），则 iPtr++ 等于 iPtr + 1 * 1 = 0x00000101

5.4 一维数组与指针

5.4.1 数组名是指向数组第一个元素的指针

数组名代表数组首元素的地址，即 *数组名是指向数组第一个元素的指针*。

例如，对于一个数组 a[10] 来说，数组名 a 代表数组 a[10] 中第一个元素 a[0] 的地址，即 a 与 &a[0] 等价。

注意：数组名 a 是地址常量，不是变量，不能给 a 赋值。

5.4.2 利用指针变量引用数组元素

若定义数组 int a[10]; 指针 int *pointer; 则 pointer = a; 等价于 pointer = &a[0];

访问数组时，pointer + i 等价于 a + i，也等价于 &a[i]。*(pointer + i) 等价于 *(a + i)，也等价于 a[i]。

在表示形式上，pointer[i] 等价于 *(pointer + i)。

需要注意的问题：

- int *p = &a[0];
  - a++ 是没有意义的，但 p++ 会引起 p 变化。
  - p 可以指向数组最后一个元素以后的元素。
- 指针做加减运算时一定要注意有效的范围

```
int a[5];
int *iPtr = &a[1];
iPtr--;      // now iPtr points to a[0]
*iPtr = 3;   // a[0] = 3
iPtr--;      // now iPtr points to a[-1], dangerous
*iPtr = 6;   // damage
```

- 指针与 ++/- - 的优先级问题

  若定义 int a[5] = {1, 2, 3, 4, 5}; int *p; 设当前 i = 3; a[i] = 4; 则
  - *++p 相当于 a[++i]，先将 p 自加，再做 * 运算
  - *--p 相当于 a[--i]，先使 p 自减，再做 * 运算
  - *p++ 相当于 a[i++]，先做 * 运算，再将 p 自加
  - *p-- 相当于 a[i--]，先做 * 运算，再将 p 自减

5.4.3 C 语言标准中关于数组的一些说明

Types

An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called "array of T". The construction of an array type from an element type is called "array type derivation".

Lvalues[1], arrays, and function designators

Except when it is the operand of the `sizeof` operator, the `_Alignof` operator, or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type of "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

Array subscripting

A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is indentical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the inital element of an array object) and `E2` is an integer. `E1[E2]` designates the `E2`-th element of `E1` (counting from zero).

例如定义了一个数组 `int a[4] = {1, 3, 5, 7};` 则数组名 `a` 相当于指向数组第一个元素 `a[0]` 的指针，即 `a` 相当于 `&a[0]`。

- `&a` 相当于管辖范围"上升"了一级
  - `&a` 是"指向数组"的指针；`&a + 1` 将跨越 16 个字节
- `*a` 相当于管辖范围"下降"了一级
  - `*a` 是数组的第一个元素 `a[0]`；即 `*a` 等价于 `a[0]`

5.5 二维数组与指针

定义 `int a[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};`

二维数组 `a[3][4]` 包含三个元素：`a[0]`，`a[1]`，`a[2]`。每个元素都是一个"包含四个整型元素"的一维数组。二维数组数组的第一个元素是 `a[0]`，`a[0]` 是一个"包含四个整型元素"的一维数组。

二维数组 `a[3][4]` 的数组名 `a` 是指向数组第一个元素 `a[0]` 这个小数组的指针。而 `a[0]` 的含义是指向第一个小数组中第一个元素 `a[0][0]` 的指针。

推论：

[1] Lvalues and Rvalues (Visual C++), https://msdn.microsoft.com/en-us/library/f90831hc.aspx

- a 与 &a[0] 等价
- a[0] 与 &a[0][0] 等价
- a[0] 与 *a 等价
- a[0][0] 与 **a 等价

三条规律：

- 数组名相当于指向数组第一个元素的指针
- &E 相当于把 E 的管辖范围上升了一个级别
- *E 相当于把 E 的管辖范围下降了一个级别

5.6 指针与函数

5.6.1 指针变量做函数参数

示例代码：

```
void Rank(int *q1, int *q2)
{
    int temp;
    if (*q1 < *q2) {
        temp = *q1;
        *q1 = *q2;
        *q2 = temp;
    }
}

int main(int argc, char const *argv[]) {
    int a, b, *p1, *p2;
    cin >> a >> b;
    p1 = &a;
    p2 = &b;
    Rank(p1, p2);
    cout << a << " " << b << endl;
    return 0;
}
```

之前学过的是，调用函数时，它会把是实参的值复制一份给形参。在上面的示例中，调用 Rank(int *q1, int *q2) 函数，p1 和 p2 的值会复制给 q1 和 q2，但由于 p1、p2 中存储的分别是变量 a 和 b 的地址，所以这之后 q1 和 q2 会分别指向变量 a 和 b。

5.6.2 数组名作为函数参数

示例代码：

```
void sum(int *p, int n)
{
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += *p++;
    }
    cout << total << endl;
}

int main(int argc, char const *argv[]) {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    sum(a, 10);
    return 0;
}
```

通过上面的示例可以看出，可以将数组名作为实参，赋值给指针类型的形参。下面是多维数组名做函数参数的情况。

示例代码：

```
int maxValue(int (*p)[4])
{
    int max = p[0][0];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++) {
            if (p[i][j] > max) {
                max = p[i][j];
            }
        }
    }
    return max;
}

int main(int argc, char const *argv[]) {
    int a[3][4] = {{1, 3, 5, 7}, {9, 11, 13, 15}, {2, 4, 6, 8}};
    cout << "The max value is " << maxValue(a) << endl;
    return 0;
}
```

在多维数组作为函数参数时，函数的形参写作指针比较麻烦，例如上面的 `(*p)[4]`。这时可以考虑将“数组名”作为形参。

示例代码：

```
// Sum all elements to the last element, but will change the array.
int sum(int array[], int n)
{
    for (int i = 0; i < 10 - 1; i++) {
        *(array + 1) = *array + *(array + 1);
        array++;    // Attention!
    }
    return *array;
}

int main(int argc, char const *argv[]) {
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << sum(a, 10) << endl;
    return 0;
}
```

前面学到数组名不是变量，不能进行 ++/- - 操作，但是上面的示例是可以正确运行的。因为 C++ 编译器会将形参数组名作为指针变量来处理。

在编写程序时，将变量地址或数组名传递给一个函数是很危险的，需要谨慎操作，必要时可以对指针的功能加以限制。

5.6.3 如何“限制”指针的功能

符号常量声明语句：

- 方式一：const 数据类型常量名 = 常量值;
- 方式二：数据类型 const 常量名 = 常量值;

定义一个指向符号常量的指针：`const int *p;`
示例代码：

```
int a = 256;
const int *p = &a;
*p = 257;   // error: read-only variable is not assignable
cout << *p << endl;
```

在上面的代码中，把 p 定义为了一个指向符号常量的指针，它所指向的内容是不能被修改的。

在指针或数组名作为函数形参时，如果不希望函数改变指针所指向的内容或数组的内容，可以在定义时在该形参前加 const 限定符。

如果 p 是一个指向符号常量的指针，我们就不可以修改它指向的内容，但我们可以改变 p 本身的值，使它指向其他内容。

示例代码：

```
const int c = 78;
const int d = 28;
int e = 18;
const int *pi = &c;
// *pi = 58;                // error: read-only variable is not assignable
pi = &d;                    // pi can be re-assigned value, now pi points to d
// *pi = 68;                // error: read-only variable is not assignable
pi = &e;                    // now pi points to e
// *pi = 88;                // error: read-only variable is not assignable
```

5.6.4 指针做函数返回值

返回指针类型的函数：`int *function(int x, int y);`

示例代码：

```
int *get(int arr[][4], int n, int m)
{
    it *pt;
    pt = *(arr + n - 1) + m - 1;
    return(pt);
}

int main(int argc, char const *argv[]) {
    int a[4][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
    int *p;
    p = get(a, 2, 3);
    cout << *p << endl;
    return 0;
}
```

当我们从一个函数中返回一个指针，而指针指向的是一个临时变量（局部变量），这将是非常危险的，我们需要确保函数返回的是一个处于生命周期中的变量的地址。例如，可以返回全局变量的地址，而非局部变量的地址。但是使用全局变量会破坏程序的结构性，这时可以使用静态局部变量。

一个函数中的静态局部变量的值在函数调用结束后不消失而保留原值。即其占用的存储单元不释放，在下一次该函数调用时，仍可以继续使用该变量。其生存周期是定义该变量开始，一直到整个程序结束。跟静态局部变量相对应的是动态局部变量，也就是普通的局部变量，它们的有效范围是从定义该变量开始到函数结尾。

使用关键字 static 定义静态局部变量：`static int value = 20;`

示例代码：

```
void fun()
{
    int a = 0;
    static int b = 0;
```

```
    a = a + 1;
    b = b + 1;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}

int main(int argc, char const *argv[]) {
    for (int i = 0; i < 5; i++) {
        fun();
        cout << "Call again!" << endl;
    }
    return 0;
}
```

运行上面的代码后，最后一次输出 a = 1、b = 5。虽然 fun() 函数被调用了多次，但 static int b = 0; 这个定义静态局部变量 b 的语句仅被执行了一次。

6 结构体与链表

6.1 结构体

我们经常希望用一组变量来描述同一个“事物”，可以将这一组变量“捆绑”起来构造一个新的数据类型，称为结构体。

示例代码：

```
struct student {
    int id;
    char name[20];
    char sex;
    int age;
    int score;
    char address[30];
};  // 注意大括号后的“;”
```

结构体一种数据类型，如果要使用结构体，就要使用它来定义变量。

6.1.1 定义结构体变量的方式

1. 直接用已生命的结构体类型定义变量名：
   `student student1, student2;`
2. 在声明类型的同时定义变量：

```
struct student {
    int id;
    char name[20];
    char sex;
    int age;
    int score;
    char address[30];
} student1, student2;    // 注意最后的“;”
```

6.1.2 结构体变量的初始化和赋值

结构体变量的初始化和数组的初始化很相似：

```
struct student {
    int id_num;
    char name[10];
};

student mike = {123, {'m', 'i', 'k', 'e', '\0'}};
```

赋值时，相当于把一个结构体变量的值复制了一份给另一个结构体变量。

6.1.3 结构体作为函数参数、函数返回值

结构体变量名作为函数参数时，是把结构体变量复制了一份给函数的形参，这与数组名作为函数参数时不同。

结构体变量作为函数返回值时，相当于复制了一份给调用者。

6.1.4 指向结构体的指针

定义指向结构体变量的指针：

```
student mike = {123, {'m', 'i', 'k', 'e', '\0'}};
student *one = &mike;
cout << (*one).id_num << " " << (*one).name << endl;     // Right.
cout << one->id_num << " " << one->name;                 // Right. '->' 为指向运算符

// error: member reference type 'student' is not a pointer;
// cout << mike->id_num << " " << mike->name;
```

指向结构体的指针作为函数参数时，函数就拿到了结构体的地址，可以改变结构体的内容，这和普通类型变量的情况是一致的。

6.1.5 结构体数组

结构体数组和普通类型的数组也是一样的，数组名是指向数组首元素的指针：

```
student myClass[3] = {
    123, {'m', 'i', 'k', 'e', '\0'},
    133, {'t', 'o', 'm', '\0'},
    143, {'j', 'a', 'c', 'k', '\0'}
};
student *one = myClass;
cout << one->id_num << " " << one->name << endl;    // 123 mike
one++;
cout << one->id_num << " " << one->name << endl;    // tom
```

综上，结构体数据类型的特性与普通数据类型的特性是一致的。

6.2 链表

在使用结构体数组时会有一些缺点，例如我们想在数组中间插入一个结构体时，为了保持结构体的次序，我们必须把这之后结构体整体向后移动。而链表这种数据结构可以很好地解决这个问题，使得各个结构体之间的连接非常灵活。

6.2.1 链表的结构

- 链表头：指向第一个链表结点的指针
- 链表结点：链表中的每一个元素，包括：
  - 当前节点的数据
  - 下一个结点的地址
- 链表尾：不再指向其他结点的结点，其地址部分放一个 NULL，表示链表到此结束。

6.2.2 动态创建链表

动态地申请内存空间：

```
// 申请一块内存空间来存放一个 int，初始值为 1024，并返回指向该内存空间的指针 pint
int *pint = new int(1024);
// 释放 pint 指向的内存
delete pint;

// 申请一块内存用来存放大小为 4 的 int 型数组，并返回指向该数组的指针 pia
int *pia = new int[4];
// 释放 pia 所指向的数组的内存
delete[] pia;
```

在创建链表和对链表进行操作时，经常需要用到 new 来动态分配内存。什么时候需要用 new，什么时候不需要用呢？这主要是看我们是否真的需要一块新的内存。有时候只是新建一个指针 temp 就能解决问题，并不需要申请内存空间。

动态地创建链表结点：

```
struct student {
    int id;
    student *next;
};

student *head;
head = new student;
```

创建链表的步骤：

1. 建立头结点 head，新建临时的指针 temp 也指向 head，用于创建后续结点。

   ```
   head = new student;
   student *temp = head;
   ```

2. 判断是否还要创建结点。

   - 如果需要创建新结点，新申请内存创建一个新的结点，并利用 temp 指针将前一个结点的 next 指针指向新结点，最后将 temp 指向新结点，以便于以后再创建新结点。

     ```
     temp->next = new student;
     temp = temp->next;
     ```

     创建完成后，返回步骤 2。

   - 如果不需要再创建结点，执行步骤 3。

3. 将最后一个结点的 next 指针置为 NULL。

   ```
   temp->next = NULL;
   ```

6.2.3 链表的操作

在链表创建完成后，我们可能需要对链表进行各种操作，包括遍历链表元素、删除结点、插入结点等。

链表元素的遍历，即依次访问链表的所有元素：

```
student *pointer = create();
while (pointer->next != NULL) {
    cout << pointer->id << endl;
    pointer = pointer->next;
}
```

删除链表中的结点时，需要分不同的情况进行讨论：

- 删除头结点

  ```
  temp = head;
  head = head->next;
  delete temp;
  ```

- 删除中间结点和末尾结点

```
follow->next = temp->next;
delete temp;
```

在链表中插入一个结点也需要分情况讨论。在向链表中插入新结点时需要格外小心，因为插入结点的操作有严格的顺序，必须先让新结点指向后续结点，然后再让前一结点指向新结点。反之，后续结点将会丢失。例如将结点 unit 插入链表：

- 在最前面插入结点

```
unit->next = head;
head = unit;
```

- 在中间插入结点

```
unit->next = temp;
follow->next = unit;
```

- 在末尾插入结点

```
temp->next = unit;
unit->next = NULL;
```

6.2.4 双向链表

前面讨论的链表都是单向链表，在这样的链表中，每个结点只包含一个指向其后继结点的指针。除了单向链表外，双向链表也很常用。在双向链表中，每个结点除了数据区域外，还包含两个指针（next、ahead），分别指向其后继结点和前趋结点。

由于涉及两个指针域，双向链表的删除结点和插入结点操作要比单向链表稍复杂一些。

删除双向链表的 temp 结点：

```
temp->ahead->next = temp->next;
temp->next->ahead = temp->ahead;
```

在双向链表中 temp 结点之后插入 unit 结点：

```
unit->next = temp->next;
unit->ahead = temp->ahead;
temp->next->ahead = unit;
temp->next = unit;
```

7 轻叩面向对象之门

7.1 再谈什么是计算机程序

计算机程序是现实世界的解决方案在计算机系统中的映射。程序开发的本质就是将现实世界的解决方案映射到计算机系统中，而它依赖的工具是编程语言。

7.2 为什么要面向对象

如果你客观地观察这个世界，你就会发现它是如此的简单，以至于像我们所想象的那样。

—— 爱因斯坦

现实世界的解决方案中包含很多事物，某些事物表现为一些数据，某些事物还包含对数据的一些动作。在描述解决方案时，可以将其中所包含的各种事物作为核心，把整个系统的行为描述为各种事物所具有的数据、以及各种事物对这些数据进行操作的动作的集合。在面向对象的方法中，将系统中的各个事物称为对象，每个对象既可以包含数据也可以执行某些动作，从而就可以把现实世界中的解决方案分解为各个对象所有的数据以及动作的集合。按照这种方法所设计的用于描述解决方案的语言就称为面向对象的语言。

接下来的问题：

- 怎么理解软件的本质?
- 什么是对象?
- 什么是面向对象的程序设计?
- 面向对象语言的作用?
- 如何通过面向对象的视角来观察世界?
- 如何进行面向对象的分析?
- 如何进行面向对象的程序设计?
- ……